Hypothetical Datalog: Negation and Linear Recursion*

Anthony J. Bonner
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903

bonner@db.toronto.edu

This paper appears in the *Proceedings of the ACM Symposium on the Principles of Database Systems (PODS)*, pages 286–300. Conference held March 1989 in Philadelphia, PA.

Abstract

This paper examines an extension of Horn logic in which rules can add entries to a database hypothetically. Several researchers have developed logical systems along these lines, but the complexity and expressibility of such logics is only now being explored. It has been shown, for instance, that the data-complexity of these logics is $PSPACE$-complete in the function-free, predicate case. This paper extends this line of research by developing syntactic restrictions with lower complexity. These restrictions are based on two ideas from Horn-clause logic: *linear recursion* and *stratified negation*. In particular, a notion of stratification is developed in which negation-as-failure alternates with linear recursion. The complexity of such rulebases depends on the number of layers of stratification. The result is a hierarchy of syntactic classes which corresponds exactly to the polynomial-time hierarchy of complexity classes. In particular, rulebases with k strata are data-complete for Σ_k^P. Furthermore, these rulebases provide a complete characterization of the relational queries in Σ_k^P. That is, any query whose graph is in Σ_k^P can be represented as a set of hypothetical rules with k strata. Unlike other expressibility results in the literature, this result does not require the data domain to be linearly ordered.

*Published in the *Proceedings of the Eighth ACM Symposium on Principles of Database Systems (PODS)*, pages 286–300, Philadelphia, Pennsylvania, March 29-31, 1989. © 1989 ACM

1 Introduction

This paper examines an extension of Horn-clause logic in which rules can add facts to a database hypothetically. Several researchers in the logic-programming community have pointed out the utility of such rules and have developed systems along these lines. Miller, for instance, has shown how such rules can structure the runtime environment of a logic program [19], and Warren and Manchanda have proposed such logics for reasoning about database updates [23, 15]. The legal domain has inspired much work into this kind of hypothetical reasoning. Gabbay, for example, has reported a need to augment Prolog with hypothetical rules in order to encode the British Nationality Act. The act contains rules such as, "You are eligible for citizenship if your father would be eligible if he were still alive" [9]. In addition, McCarty has developed a wide class of hypothetical rules for the purpose of constructing computer-based legal consultation systems, especially systems for reasoning about contract law and corporate tax law [16, 18].

Although hypothetical reasoning is complex in general [11], these systems focus on a form of hypothetical reasoning which appears tractable. In particular, they augment Horn-clause logic with rules of the form $A \leftarrow B[add : C]$ which intuitively means, "infer A if inserting C allows the inference of B." The formal properties of these rules are still being explored. Gabbay has shown that they have an intuitionistic semantics [8], and Miller has developed fixpoint semantics for the predicate case [19]. McCarty has extended this work to a larger class of formulas and established interesting semantic results [16, 17]. Bonner has shown that query evaluation in such systems is data-complete for $PSPACE$ in the function-free predicate case (data-complete for $EXPTIME$ when hypothetical deletions are allowed) [4]. This paper extends this line of research by developing syntactic restrictions whose data-complexity is less that $PSPACE$.

Central to these restrictions is the idea of linearity. A rule is linear if recursion occurs through only one premise. In Horn-clause logic, "linear rules play an important role because, (i) there is a belief that most 'real life' recursive rules are linear, and (ii) algorithms have been developed to handle them efficiently" [2]. Linearity, however, does not affect the data-complexity of Horn-clause rulebases, even when combined with negation-by-failure. In each case, the data-complexity is simply P. For hypothetical rules, the situation is more interesting. Firstly, linearity reduces their data-complexity from $PSPACE$ to NP. Secondly, when negation-by-failure is introduced, data-complexity is determined by the interaction between linear recursion and negation.

To capture this, we develop a new notion of stratification, in which linear recursion alternates with negation-as-failure. The complexity of a hypothetical rulebase then depends on its degree of stratification. As the number of strata increases, the data-complexity climbs the polynomial-time hierarchy [21]. In particular, rulebases with k strata are data-complete for Σ_k^P.

The polynomial-time hierarchy is a sequence of complexity classes between P and $PSPACE$. It is based on the idea of an oracle Turing-machine[12] and for our purposes, can be defined recursively as follows:

- $\Sigma_1^P = NP$
- $\Sigma_{k+1}^P = NP^{\Sigma_k^P}$ = Those languages accepted in non-deterministic polynomial time by an oracle machine whose oracle is a language in Σ_k^P.

For all k, $\quad P \subseteq \Sigma_k^P \subseteq \Sigma_{k+1}^P \subseteq PSPACE$.[1]

This paper also addresses the issue of expressibility. A central result is that hypothetical rulebases with k strata provide a complete characterization of the relational queries in Σ_k^P. That is, any relational query whose graph is in Σ_k^P can be represented as a hypothetical rulebase with at most k strata. The proof relies on a simulation of oracle Turing machines, and in this respect, is similar to other expressibility proofs in the literature [13, 22]. One difference, however, is that we do not require the data domain to be linearly ordered. Linearly-ordered domains are used to simulate counters, which in turn, are used to simulate tape-head movements. Our approach is to start with unordered domains and assert linear orders hypothetically. This technique works for all generic queries, that is, for all queries satisfying the *consistency criterion* of Chandra and Harel [6, 7].

2 Examples

This section gives several examples of hypothetical queries and rules. In each example, tuples are hypothetically inserted into the database before a least-fixpoint query is made. The queries are expressed with a modal-like operator $Q[add : P]$ which means, "if P were inserted in the database, then Q could be inferred." The notation $R, DB \vdash \phi$ means that the formula ϕ can be inferred from the rulebase R and the database DB.

The examples are centered on a rule-based system which describes university policy. For instance, the atomic formula $take(s, c)$ intuitively means that student s has taken course c, and $grad(s)$ means that s is eligible for graduation. The database DB contains facts such as $take(tony, cs250)$, and the rulebase R contains rules such as

$$grad(s) \leftarrow take(s, his101), take(s, eng201).$$

In the following examples, each query is described in three ways: (i) informally in English, (ii) formally at the meta-level, and (iii) formally at the object-level with operators of addition.[2]

[1] Although considered likely, it is an open question as to whether these containments are strict.

[2] See [14] for a description of meta-level and object-level reasoning.

Example 1. Consider the query, "If Tony took $cs452$, would he be eligible to graduate?" That is, if $take(tony, cs452)$ were added to the database, could we infer $grad(tony)$? This query can be formalized at the meta-level as follows:

$$R, DB + take(tony, cs452) \vdash grad(tony)$$

In our language of hypotheticals, the expression $grad(tony)[add : take(tony, cs452)]$ represents this query. It is an object-level level expression ψ such that $R, DB \vdash \psi$ iff the meta-level condition is satisfied.

Example 2. "Retrieve those students who could graduate if they took one more course." i.e., at the meta-level, we want those s such that

$$\exists c\ [R, DB + take(s, c) \vdash grad(s)]$$

At the object-level, the expression $\psi(s) = \exists c, grad(s)[add : take(s, c)]$ represents this query. That is, for each value of s, $R, DB \vdash \psi(s)$ iff the meta-level condition is satisfied.

Having introduced hypothetical queries, we can use them in the premises of rules. These rules have the form $A \leftarrow B[add : C]$ which intuitively means, "infer A if inserting C into the database allows the inference of B." Such rules turn the query language into a logic for building rulebases.

Example 3. Suppose the university has the following policy:

> "A student qualifies for a degree in *math and physics* if he is within one course of a degree in *math* and within one course of a degree in *physics*."

This policy can be expressed with two rules:

$$within1(s, d) \leftarrow \exists c\ grad(s, d)[add : take(s, c)].$$
$$grad(s, math\&phys) \leftarrow within1(s, math),\ within1(s, phys).$$

Here, $grad(s, d)$ means that student s is eligible for a degree in discipline d, and $within1(s, d)$ means that s is within one course of a degree in d. Note that the premise of the first rule is a hypothetical query similar to the one in example 2. [3] shows that there is a strong sense in which such rules cannot be expressed in Datalog.

3 Hypothetical Inference

This section defines a logical inference system for hypothetical rules and queries.[3] It is an extension of Horn logic, both syntactically and proof theoretically, and some of the terminology is borrowed from first-order predicate logic. In this paper, all formulas are are function-free.

[3] See [3] for a proof that this inference system has an intuitionistic semantics.

Definition 1 *A premise (or query) is an expression having one of the following forms:*

- A *where A is an atomic formula.*
- $A[add: B]$ *where A and B are atomic formulas.*

Definition 2 *A hypothetical rule is an expression of the form* $A \leftarrow \phi_1, \phi_2, ..., \phi_k$ *where $k \geq 0$, A is atomic, and each ϕ_i is a premise.*

Definition 3 *Suppose R is a set of hypothetical rules and DB is a database. Then hypothetical inference is defined as follows, where A and B are ground atomic formulas:*

1. $R, DB \vdash A$ *if* $A \in DB$
2. $R, DB \vdash A[add: B]$ *if* $R, DB + \{B\} \vdash A$
3. $R, DB \vdash A$ *if for some rule* $A' \leftarrow \phi_1, ..., \phi_k$ *in R, and for some ground substitution θ over $dom(R, DB)$, it is the case that* $A = A'\theta$ *and* $R, DB \vdash \phi_i\theta$ *for each i.*

Here, $dom(R, DB)$ denotes the domain of the rulebase R and the database DB. It includes all constant symbols appearing in R and DB.

Example 4. Suppose the rulebase R consists of Horn rules defining a predicate D plus the following $n + 1$ rules:

$$
\begin{aligned}
&A_1 \leftarrow A_2[add: B_1] \\
&A_2 \leftarrow A_3[add: B_2] \\
&\quad \cdots \\
&A_n \leftarrow A_{n+1}[add: B_n] \\
&A_{n+1} \leftarrow D
\end{aligned}
$$

Then $R, DB \vdash A_i$ iff $R, DB + \{B_i, ..., B_n\} \vdash D$.

Example 5. Suppose the database DB contains the following atomic formulas, which define a linear order $a_1, a_2, ..., a_n$:

$$
\begin{gathered}
FIRST(a_1),\ NEXT(a_1, a_2),\ NEXT(a_2, a_3), \\
...\ NEXT(a_{n-1}, a_n),\ LAST(a_n).
\end{gathered}
$$

Suppose also that the rulebase R consists of rules defining the predicate D plus the following three rules:

$$
\begin{aligned}
&A \leftarrow FIRST(x),\ A'(x)[add: B(x)]. \\
&A'(x) \leftarrow NEXT(x, y),\ A'(y)[add: B(y)]. \\
&A'(x) \leftarrow LAST(x),\ D.
\end{aligned}
$$

Then $R, DB \vdash A$ iff $R, DB + \{B(a_1), ..., B(a_n)\} \vdash D$.

3.1 Negation by Failure

There are many queries which the above inference system cannot express. This is because the system is *monotonic*: as entries are added to the database, inferences do not disappear. Clearly many queries are non-monotonic, such as relational-algebra queries involving complementation. Negation-by-failure makes the inference system non-monotonic. It is defined by adding the following inference rule to those of definition 3:

$$R, DB \vdash \sim \phi \quad \textit{if} \quad R, DB \not\vdash \phi$$

Rulebases involving negation-by-failure are not always well-defined. This is a familiar problem in Horn-clause logic. For example, given the two rules $A \leftarrow \sim B$ and $B \leftarrow \sim A$, it is unclear whether A is to be inferred, or B, or both, or neither. As in the Horn-clause case, however, if there is no recursion through negation, then there is no ambiguity. In this paper, therefore, we assume that negation is stratified [1].

This paper makes one other simplifying assumption: that only atomic queries are negated. That is, $\sim A$ is allowed as a rule premise, but $\sim A[add : B]$ is not. This restriction is a theoretical convenience but poses no serious limitations in practice. One can always introduce a new predicate C and a new rule $C \leftarrow A[add : B]$, so that $\sim C$ has the same effect as $\sim A[add : B]$.

Example 6. Suppose R is the following collection of rules:

$$EVEN \leftarrow SELECT(\overline{x}),\ ODD[add : B(\overline{x})].$$
$$ODD \leftarrow SELECT(\overline{x}),\ EVEN[add : B(\overline{x})].$$
$$EVEN \leftarrow \sim SELECT(\overline{x}).$$
$$SELECT(\overline{x}) \leftarrow A(\overline{x}),\ \sim B(\overline{x}).$$

Then $R, DB \vdash EVEN$ iff DB contains an even number of entries of the form $A(\overline{x})$.

In this example, the rulebase determines the parity of a relation A. The rulebase introduces a new relation B and hypothetically copies A to B one tuple at a time. As tuples are copied to B, the first two rules "flip back and forth" between the two predicates $EVEN$ and ODD, reflecting the changing parity of the difference relation $A \sim B$. During this phase, the fourth rule selects tuples from A which are not yet in B. When A has been completely copied to B, $A \sim B$ is empty, and the third rule infers that $EVEN$ is true.

Note that it does not matter in which order the elements of A are copied to B. Every order will give the same answer: either every order will result in a proof of $EVEN$ or every order will result in a proof of ODD. This idea of order independence is an important aspect of the expressibility results in section 6.

Example 7. Suppose that DB is a database representing a directed graph. That is, $NODE(a) \in DB$ iff a is a node in the graph, and $EDGE(a,b) \in DB$ iff there is an edge in the graph from a to b. Suppose also that R is the following collection of rules:

$$YES \leftarrow NODE(x),\ PATH(x)[add : PNODE(x)].$$
$$PATH(x) \leftarrow SELECT(y),\ EDGE(x,y),\ PATH(y)[add : PNODE(y)].$$
$$PATH(x) \leftarrow \sim SELECT(y).$$
$$SELECT(y) \leftarrow NODE(y),\ \sim PNODE(y).$$

Then $R, DB \vdash YES$ iff the graph represented by DB has a directed Hamiltonian path.

In this example, the data-complexity of the rulebase is NP-hard [10]. To find a Hamiltonian path, the rulebase looks for a sequence of edges that contains each node in the graph exactly once. The first rule selects a node x at which this sequence is to begin. The second rule is then applied repeatedly, selecting a node y connected by an edge to the last node in the path. Each time a node is selected, $PNODE(y)$ is hypothetically added to the database, recording the fact that node y is in the path. This enables the fourth rule to select nodes which are not yet in the path. A Hamiltonian path has been found when it is not possible to select any more nodes, i.e., when all nodes in the graph are also in the path. The third rule detects this situation. Note that all selections are non-deterministic, so the rulebase searches non-deterministically for all possible Hamiltonian paths.

It is the ability to record facts, such as which nodes are in the path, that distinguishes our logic from (function free) Horn logic and accounts for its NP-hardness. Although negation contributes to the conceptual simplicity of the above example, it does not contribute to its computational complexity, since there are sets of hypothetical rules which are NP-hard but which are negation-free.[4] Negation does, however, play a crucial role in the next example.

Example 8. Given the rulebase R in the previous example, construct a new rulebase R' by adding to it the following rule:

$$NO \leftarrow \sim YES$$

Then $R', DB \vdash YES$ iff DB contains a Hamiltonian circuit,
and $R', DB \vdash NO$ iff DB does not contain a Hamiltonian circuit.

In this example, the data-complexity of R' is both NP- and $coNP$-hard. Thus, adding a single non-recursive rule to R has increased its complexity class from NP to $NP \cup coNP$.

[4]The proof of this is left as an exercise for the reader.

4 Linear Stratification

In [4] it was shown that the inference system of section 3 is data-complete for $PSPACE$. In this section, we develop syntactic restrictions with reduced complexity. Central to these restrictions are the ideas of *linear recursion* and *stratified negation.* In particular, a new notion of stratification is developed in which linear recursion alternates with negation-by-failure. For such rulebases, data-complexity depends on the number of layers of stratification. Rulebases with k strata are data-complete for Σ_k^P.

$PSPACE$-hardness was established in [4] by encoding the computations of alternating Turing-machines. Central to these encodings are rules of the form,

$$A \leftarrow B,\ A[add : C_1],\ A[add : C_2]\ ...\ A[add : C_n]. \tag{1}$$

These rules have two important features: (i) the premise has more than one hypothetical operator, and (ii) each of these operators is recursive. To reduce complexity, we disallow such rules. We focus instead on rules in which recursion occurs through only one premise. In Horn logic, such rules are said to be linear [2].

Example 9. The following rulebase has three strata, the i^{th} stratum defining the predicate A_i.

$$\begin{aligned}
&A_3 \leftarrow B_3,\ A_3[add : C_3] \\
&A_3 \leftarrow D_3,\ \sim A_2. \\[4pt]
&A_2 \leftarrow B_2,\ A_2[add : C_2] \\
&A_2 \leftarrow D_2,\ \sim A_1. \\[4pt]
&A_1 \leftarrow B_1,\ A_1[add : C_1] \\
&A_1 \leftarrow D_1
\end{aligned}$$

By alternating linear recursion with negation-by-failure, stratified rulebases can be built, as in example 9 above. The definitions below define stratification precisely. They are are generalizations of those given in [1] and [2]. The essential idea is to partition a rulebase into segments, numbered $1, ..., n$. Even segments must contain hypothetical rules with linear recursion, and odd segments must contain Horn rules with stratified negation. A *stratum* is then defined to consist of two adjacent segments, one odd and one even.

Definition 4

- *A predicate symbol B occurs positively in a rule iff there is a formula of the form $B(\overline{x})$ in the premise of the rule.*

- *A predicate symbol B occurs negatively in a rule iff there is a formula of the form $\sim B(\overline{x})$ in the premise of the rule.*
- *A predicate symbol B occurs hypothetically in a rule iff there is a formula of the form $B(\overline{x})[add : C(\overline{y})]$ in the premise of the rule.*

Definition 5 *Suppose R is a set of hypothetical rules and A is a predicate symbol. Then the definition of A is the set of rules in R whose conclusion is a formula of the form $A(\overline{x})$.*

Definition 6 *(H-stratification) A set of hypothetical rules R is H-stratified if there is a partition $R = R_1 \cup R_2 \cup \cdots R_n$ such that*

- *If a predicate symbol occurs positively in a rule of R_i, then its definition is contained in $\cup_{j \leq i} R_j$.*
- *If a predicate symbol occurs negatively in a rule of R_{2i}, then its definition is contained in $\cup_{j<2i} R_j$.*
- *If a predicate symbol occurs hypothetically in a rule of R_{2i-1}, then its definition is contained in $\cup_{j<2i-1} R_j$.*

Definition 7 *(Strata) Suppose R is an H-stratified rulebase with partition $R_1, R_2 ... R_n$. Then $\Delta_i = R_{2i-1}$ and $\Sigma_i = R_{2i}$. i.e., $R = \cup_{i \geq 1}(\Delta_i \cup \Sigma_i)$. We call $\Delta_i \cup \Sigma_i$ the i^{th} stratum of R.*

Each stratum thus has two parts: (i) an upper part Σ_i consisting mostly of hypothetical rules (without negation), and (ii) a lower part Δ_i consisting mostly of Horn rules (with negation). In general, Σ_i or Δ_i may be empty.

Example 10. The following rulebase is H-stratified and has two strata. (Δ_1 is empty.)

$$\begin{array}{ll}
\Sigma_2 & \left\{ \begin{array}{l} A_2 \leftarrow A_2[add : E_2],\ A_2[add : F_2]. \\ A_2 \leftarrow \sim B_2. \end{array} \right. \\
\Delta_2 & \left\{ \begin{array}{l} B_2 \leftarrow \sim C_2,\ B_2. \\ C_2 \leftarrow \sim D_2,\ C_2. \\ D_2 \leftarrow A_1[add : G_1]. \end{array} \right. \\
\Sigma_1 & \left\{ \begin{array}{l} A_1 \leftarrow A_1[add : E_1]. \\ A_1 \leftarrow A_1[add : F_1]. \\ A_1 \leftarrow \sim B_1. \end{array} \right.
\end{array}$$

In H-stratified rulebases, hypothetical rules alternate with negation-as-failure. H-stratification, however, does not exclude recursion through negation, nor does it exclude rules of the form (1), as example 10 illustrates. A stronger notion is therefore required, one which we call *linear stratification*, or stratification for short. In a rulebase with linear stratification, each Δ_i is defined to have stratified negation, and each Σ_i is defined to have linear recursion.

To guarantee linearity, it is not enough that no rule have the form (1). For instance, each of the following $n+1$ rules may appear linear, but taken together, they imply rule (1):

$$A \leftarrow B,\ D_1 ... D_n.$$
$$D_i \leftarrow A_i[add : C_i]$$

Linearity is defined precisely for Horn rules in [2], and is easily extended to include hypothetical rules. It is based on the idea of mutually recursive predicates. (e.g., in example 6, the predicates $EVEN$ and ODD are mutually recursive.) Linearity permits mutual recursion as long as the recursion is not equivalent to a rule of the form (1).

Definition 8 *(Linearity) A hypothetical rule* $B \leftarrow \phi_1, ..., \phi_n$ *is linear if the premise has exactly one occurrence of a predicate which is mutually recursive with* B*. A rule is recursive if there is at least one such occurrence. A set of rules is linear if every recursive rule is linear.*

Definition 9 *(Linear Stratification) A set of hypothetical rules is linearly stratified if it is H-stratified and it satisfies the following two conditions:*

- *Each* Σ_i *is linear.*
- *Each* Δ_i *has stratified negation.*

In example 9, the rulebase is linearly stratified. In example 10, it is not.

Lemma 1 *Given a set* R *of hypothetical rules, determining whether* R *is linearly stratified is decidable in polynomial time (polynomial in the size of the rulebase). If* R *is linearly stratified, then* Σ_i *and* Δ_i *can be computed in polynomial time for some stratification of* R*.*

To determine whether a rulebase is linearly stratifiable is straightforward. First compute all the equivalence classes of mutually-recursive predicates. If any equivalence class has recursion through negation, then fail. If any equivalence class has both hypothetical recursion and non-linear recursion, then fail. The rulebase is linearly stratifiable iff neither test fails. The first test guarantees that negation within each Δ_i is stratified; the second test guarantees that recursion within each Σ_i is linear.

To actually generate a linear stratification, we use a relaxation algorithm. When the algorithm terminates, each predicate symbol P is assigned a partition number $part(P)$, as in definition 6. From this, definition 7 determines to what stratum each predicate belongs. Initially, each predicate is assigned to partition number 1; then the following procedure is executed.

```
do until the partition numbers stop changing:
    do for each predicate symbol P:
        if part(P) does not satisfy definition 6
        then increment part(P) by 1;
end;
```

On every iteration of the outer loop, except the last, the partition number of some predicate symbol must increase by 1. Since the rulebase is guaranteed to be linearly stratified, this will continue until, at worst, each predicate is assigned to a unique partition. i.e., the number of iterations of the outer loop is $O(m^2)$, where m is the number of predicate symbols.

5 Data-Complexity

Data-complexity is the complexity of evaluating a query as a function of database size. It is defined precisely in terms of the *graph* of a database query [22].

Definition 10 *Suppose that ψ is a relational database query. The graph of ψ is the set of ordered pairs $(\overline{x}, DB)$ where $\overline{x}$ is an answer to the query when applied to database DB.*

The *data complexity* of a set of queries is the complexity of their graphs. In particular,

Definition 11 *A set of queries is data-complete in a complexity class $\mathcal{C}$ if (1) the graph of each query is in $\mathcal{C}$, and (2) there is some query in the set whose graph is a complete language for $\mathcal{C}$.*

This section views a rulebase as a database query and establishes the data-complexity of rulebases with linear stratification. In particular,

Theorem 1 *Let $\mathcal{R}_k$ be the set of hypothetical rulebases having at most k levels of linear stratification. $\mathcal{R}_k$ is data-complete for Σ_k^P.*

The proof is presented in the following two sections.

5.1 Lower Complexity Bound

This section shows that the data-complexity of a rulebase having k strata is Σ_k^P-hard. Our strategy is to encode the computations of a collection of NP oracle-machines as a stratified rulebase. In particular, we construct a rulebase in which each stratum encodes a single NP oracle-machine, with each machine using the stratum below as its oracle. The k strata thus encode a cascade of k distinct NP oracle-machines, i.e., a Σ_k^P-machine. In this way, given any language L in Σ_k^P, we can encode the computations of a machine which accepts L. Besides providing a lower complexity bound, these encodings are central to the expressiveness results of section 6.

Suppose that L is a language in Σ_k^P and $\overline{s}$ is a string. We encode $\overline{s}$ as a database $DB(\overline{s})$ and construct a rulebase $R(L)$ with k strata so that

$$R(L), DB(\overline{s}) \vdash ACCEPT \quad \textit{iff} \quad \overline{s} \in L \tag{2}$$

where $ACCEPT$ is a 0-ary predicate. The important point is that the rulebase $R(L)$ is independent of the string $\overline{s}$. This allows us to infer that the data-complexity of $R(L)$ is Σ_k^P-hard. In particular, let L be a language which is Σ_k^P-complete.[5] The hardness result then follows immediately. The rest of this section describes the construction of $DB(\overline{s})$ and $R(L)$.

5.1.1 Building the Database $DB(\overline{s})$

Since $L \in \Sigma_k^P$, there is a Σ_k^P-machine which accepts it. That is, there is a sequence of NP-machines $M_k, ..., M_1$ such that (i) M_{i-1} is an oracle for M_i, and (ii) M_k accepts L. Since these machines all run in (non-deterministic) polynomial time, there is an integer l such that each machine runs in time n^l, where n is the length of the input $\overline{s}$.[6] A counter is therefore needed to represent n^l points in time and n^l positions on tape. Such a counter is easily encoded by placing the following entries into the database $DB(\overline{s})$:

$$\begin{array}{l} FIRST(0),\ NEXT(0,1),\ NEXT(1,2), \\ \qquad ...\ NEXT(n^l-2, n^l-1),\ LAST(n^l-1). \end{array}$$

Given this counter, we can represent the oracle-machines $M_1, ..., M_k$. Each machine has two tape-heads, one for reading and writing on its work tape, and one for writing onto its oracle tape (the work tape of M_{i-1}). We use the following predicates to represent machine M_i. There is one predicate for each symbol c in its tape alphabet and one for each state q in its finite control.

- $CELL_i^c(j,t)$ is true iff at time t, the work tape of M_i has symbol c in cell j.
- $CONTROL_i^q(j_1, j_2, t)$ is true iff at time t, the finite control of M_i is in state q, the work head is over cell j_1 of the work tape, and the oracle head is over cell j_2 of the oracle tape.

For each value of t, these atomic formulas define k distinct id's, one for each machine.

The database $DB(\overline{s})$ describes the initial tape contents of these machines. Since machines $M_{k-1}, ..., M_1$ act as oracles, their work tapes are initially blank. Thus, the following entries are

[5] See [7] for examples of such languages.

[6] Note that each machine can only provide its oracle with a string of polynomial length, so that the oracle runs in polynomial time, both in terms of its own input and the input of the machine which invoked it.

placed in $DB(\overline{s})$:

$$CELL_i^b(0,0),\ CELL_i^b(1,0),\ ...\ CELL_i^b(n^l-1,0).$$

where b denotes a blank. For machine M_k, the work tape must initially contain the input string $\overline{s} = < s_0, s_1, ..., s_{n-1} >$. That is, the symbol s_j must appear in cell j, and the rest must be blank. Thus, the following entries are placed in $DB(\overline{s})$:

$$\begin{array}{ll} CELL_k^{s_0}(0,0), & CELL_k^b(n,0), \\ CELL_k^{s_1}(1,0), & CELL_k^b(n+1,0), \\ \quad\cdots & \quad\cdots \\ CELL_k^{s_{n-1}}(n-1,0), & CELL_k^b(2n-1,0), \\ & \quad\cdots \\ & CELL_k^b(n^l-1,0). \end{array}$$

This completes the construction of the database $DB(\overline{s})$. It defines a counter from 0 to $n^l - 1$ and specifies the initial contents of each machine tape. Note that this construction takes polynomial time and space (polynomial in the length of $\overline{s}$).

5.1.2 Building the Rulebase $R(L)$

$R(L)$ is a stratified rulebase in which the i^{th} stratum encodes machine M_i. As an NP machine, M_i has many computation paths.[7] During inference, each computation path is "grown" one *id* at a time. In particular, the last *id* in a computation path is extended by hypothetically inserting each of its successor *id*'s into the database, one at a time. This process can be viewed as a recursive depth-first search, in which each computation path is extended one *id* at a time, tested, and then retracted. Thus the rules in the i^{th} stratum generate and test each computation path of machine M_i.

Whereas each stratum simulates a single machine M_i, the complete rulebase $R(L)$ simulates the composite machine formed from $M_k, ..., M_1$. At any point during inference, therefore, the database may contain many computation paths, one for each machine. In particular, when the composite machine has invoked oracles to a depth of j, the machines $M_k, ..., M_{k-j}$ are running. In our logical simulation of this, the database would contain $j+1$ computation paths.

The main task of the rulebase $R(L)$ is to determine when a machine *id* is *accepting*. For non-deterministic machines, acceptance is defined recursively as follows: an *id* is accepting iff (i) it contains an accepting control state, or (ii) at least one of its successor *id*'s is accepting.[8] $R_i(L)$ defines a predicate $ACCEPT_i(t)$ which determines whether the *id* at the end of a computation

[7] A computation path is a sequence of machine *id*'s.

[8] Thus a machine accepts its input iff its initial *id* is accepting.

path is accepting. In particular, if DB encodes computation paths for $M_k, ..., M_i$, and if the path for M_i ends at time t, then

$$R(L), DB \vdash ACCEPT_i(t) \quad \text{iff} \quad \text{the last } id \text{ in the computation path for } M_i \text{ is accepting.}$$

Thus, in the composite machine constructed from $M_k, ..., M_1$, the predicate $ACCEPT_i$ is defined for the latest *id* of the deepest oracle running.

The computations of the composite machine *begin* with M_k however. These computations are simulated when trying to infer the predicate $ACCEPT_k(0)$. The following rule initiates this simulation:

$$ACCEPT \quad \leftarrow \quad FIRST(x),\ ACCEPT_k(x)\ [add : CONTROL_k^{q_0}(x, x, x)].$$

This rule builds the initial *id* for the computation: it places the tape heads of M_k at the beginning of their respective tapes, and puts the finite control into its initial state q_0. The database then contains a single computation path, of length 1, consisting of the initial *id* for M_k. The rule then tries to prove $ACCEPT_k(0)$. This succeeds iff the initial *id* is accepting, i.e., iff M_k accepts its input.

Thus, given a rulebase defining the predicates $ACCEPT_i$, we can define the predicate $ACCEPT$, which in turn establishes formula (2).

5.1.3 Defining the Predicate $ACCEPT_i$

This section gives rules which define the predicate $ACCEPT_i$. There are three types of rules: (i) those which detect accepting states, (ii) those which encode the transition relation of M_i, and (iii) those which encode the mechanism for invoking the oracle M_{i-1}.

(i) Suppose that q_a is an accepting state of machine M_i. Then any *id* containing q_a is an accepting *id*. This is easily encoded with the following rule:

$$ACCEPT_i(t) \quad \leftarrow \quad CONTROL_i^{q_a}(j_1, j_2, t).$$

The variable t records the time at which acceptance occurs, and the variables j_1 and j_2 signify that the tape-head positions are unimportant.

(ii) For each element of the machine's transition relation, we write a single hypothetical rule. For example, suppose that M_i has the following transition:

> If the finite control is in state q and the work head is scanning symbol b, then (i) write symbol c onto the work tape and move the work head one cell to the left, (ii) write symbol d onto the oracle tape and move the oracle head one cell to the right, and (iii) put the finite control into state q'.

This is encoded with the following rule:[9]

$$\begin{aligned} ACCEPT_i(t) \quad \leftarrow \quad & CONTROL_i^q(j_1, j_2, t),\ CELL_i^b(j_1, t), \\ & NEXT(t, t'),\ NEXT(j_1', j_1),\ NEXT(j_2, j_2'), \\ & ACCEPT_i(t')\ [add : CELL_i^c(j_1', t'),\ CELL_{i-1}^d(j_2', t'),\ CONTROL_i^{q'}(j_1', j_2', t')]. \end{aligned}$$

This rule hypothetically inserts the updated control information into the database, thereby creating a new machine *id*.

(iii) Finally, we encode the mechanism for invoking oracles. An oracle machine has an extra tape-head, the oracle head, with which it writes a string of symbols onto the oracle's input tape. At certain points, the machine invokes the oracle, which takes its input and then replies *yes* or *no*. To carry this out, an oracle machine has three special states: $q_?$, q_y and q_n. When the machine enters the state $q_?$, the oracle is invoked and computation is suspended until the oracle returns its answer. If *yes* is returned, the machine enters state q_y; if *no* is returned, it enters state q_n.

This mechanism is encoded as two rules:

$$\begin{aligned} ACCEPT_i(t) \quad \leftarrow \quad & CONTROL_i^{q_?}(j_1, j_2, t),\ ORACLE_{i-1}(t),\ NEXT(t, t'), \\ & ACCEPT_i(t')\ [add : CONTROL_i^{q_y}(j_1, j_2, t')] \end{aligned}$$

$$\begin{aligned} ACCEPT_i(t) \quad \leftarrow \quad & CONTROL_i^{q_?}(j_1, j_2, t),\ \sim ORACLE_{i-1}(t),\ NEXT(t, t'), \\ & ACCEPT_i(t')\ [add : CONTROL_i^{q_n}(j_1, j_2, t')] \end{aligned}$$

Note the use of negation-by-failure in the second rule. This defines the boundary between one stratum of the rulebase $R(L)$ and the next. It is also the source of the Σ_k^P complexity, for without negation, oracle invocation cannot be simulated.

In these rules, the predicate $ORACLE_{i-1}$ is true iff machine M_{i-1}, acting as the oracle, returns *yes*. The following rule puts M_{i-1} into its initial configuration and begins the simulation of its computations:

$$ORACLE_{i-1}(t) \quad \leftarrow \quad FIRST(j),\ ACCEPT_{i-1}(t)\ [add : CONTROL_{i-1}^{q_0}(j, j, t)].$$

This rule places the tape heads of M_{i-1} at the beginning of their respective tapes and puts the finite control into its initial state q_0, thereby initiating computation.[10]

[9] Notice that symbol d is written onto the work tape of machine M_{i-1}.

[10] Note that until this rule is invoked, there are no formulas in the database of the form $CONTROL_{i-1}^q(j_1, j_2, t)$. This insertion thus defines a unique starting point for the computations of M_{i-1}.

5.1.4 The Frame Axiom

The above rules determine the *changes* caused to an *id* by a machine transition. The greater part of an *id*, however, remains unchanged by such transitions. Indeed, except for those cells under the tape heads, the contents of the machine tapes remain unchanged. This is an instance of the *frame axiom* [14], and we must write rules to encode it. Such rules are necessary only because we are representing time explicitly; i.e., the database represents a sequence of *id*'s, and rules are needed to copy the unchanged portion of an *id* from one instant of time to the next.

The following rules examine the work tape of machine M_i, they identify those cells *not* under a tape head at time t, and they copy their values to the same cells at time $t' = t + 1$. For each tape symbol c and each control state q, the following rules are added to the i^{th} stratum of $R(L)$.

$$\begin{array}{lcl} CELL_i^c(j,t') & \leftarrow & CELL_i^c(j,t),\ NEXT(t,t'),\ \sim HEAD_i(j,t). \\ HEAD_i(j,t) & \leftarrow & CONTROL_i^q(j,j',t). \\ HEAD_i(j,t) & \leftarrow & CONTROL_{i+1}^q(j',j,t). \end{array}$$

The predicate $HEAD(j,t)$ is true iff there is a tape head over cell j at time t, i.e., either the work head of M_i or the oracle head of M_{i+1}. Combined with those of the previous section, these rules complete the definition the rulebase $R(L)$. This in turn establishes formula (2) and hence the lower complexity bound of theorem 1.

5.2 Upper Complexity Bound

This section shows that the data-complexity of a rulebase with k strata is in Σ_k^P. To show this, we write a series of proof procedures $PROVE_k, ..., PROVE_1$, one for each stratum. Each $PROVE_i$ is a non-deterministic procedure which invokes $PROVE_{i-1}$ as a subroutine. The main part of the proof shows that if $PROVE_{i-1}$ is treated as an oracle, then $PROVE_i$ runs in non-deterministic polynomial time (polynomial in the size of the data domain). Thus each $PROVE_i$ is an NP oracle-machine, and the data-complexity of the rulebase is in Σ_k^P.

$PROVE_i$ takes two arguments, a database DB and an atomic formula $A(\overline{a})$, and returns *true* or *false*. If A is defined at or below the i_{th} stratum, then $PROVE_i[DB, A(\overline{a})] = true$ iff $R, DB \vdash A(\overline{a})$, where R is the stratified rulebase.

$PROVE_i$ is actually two procedures, $PROVE_{\Sigma_i}$ and $PROVE_{\Delta_i}$, corresponding to the two parts of the i^{th} stratum. $PROVE_{\Sigma_i}$ is an NP-machine. Operating top-down, it creates a set of subgoals. To determine whether $R, DB \vdash A(\overline{a})$, it non-deterministically chooses a rule which defines the predicate A, and places the premises of this rule in the set of subgoals. For each subgoal involving a predicate in Σ_i, the process repeats itself. Because the rules in Σ_i are linear, the set of subgoals can only grow to polynomial size. Subgoals not involving predicates defined in Σ_i are passed to $PROVE_{\Delta_i}$.

$PROVE_{\Delta_i}$ is a P-machine. Since Δ_i is essentially a set of stratified Horn rules, it can answer queries in polynomial time by generating the perfect model in a bottom-up fashion [1]. The only ruffle in this approach is that some of the rules in Δ_i may contain hypothetical premises $A[add : B]$. In such cases, however, A will be defined in a lower stratum, so $PROVE_{\Delta_i}$ invokes $PROVE_{\Sigma_{i-1}}$ as an oracle to test hypothetical premises. Aside from this, $PROVE_{\Delta_i}$ is the familiar, bottom-up procedure of stratified Horn-logic, and therefore runs in polynomial time.

The rest of this section describes these two procedures in detail for the propositional case. The predicate case is a straightforward generalization.

5.2.1 The Procedure $PROVE_{\Sigma_i}$

Suppose that ψ_0 is a formula of the form A or $A[add : B]$ where A and B are atomic and A is defined in Σ_i or below.[11] Then $PROVE_{\Sigma_i}(\psi_0, DB_0) = true$ iff $R, DB_0 \vdash \psi_0$.

Central to $PROVE_{\Sigma_i}$ is the set of subgoals which it constructs. Each of these goals is an ordered pair of the form (ψ, DB) where DB is a database and ψ has the form of a rule premise (section 3). A goal (ψ, DB) succeeds iff $R, DB \vdash \psi$.

$PROVE_{\Sigma_i}$ repeatedly chooses a goal from the goal set and expands it. This is the job of lines 1, 2 and 3 in the procedure below. These lines correspond exactly to inference rules 1, 2 and 3 in definition 3. Line 1 determines whether a goal is trivially true, and lines 2 and 3 replaces a goal by a set of subgoals.

Line 2 expands hypothetical goals into atomic goals. Atomic goals, in turn, are expanded by line 3, which chooses a rule and places its premises in the goal set. Line 3 applies only to atoms which are defined in Σ_i however. Subgoal expansion therefore stops when the goal set contains only goals of the form (A, DB) and $(\sim A, DB)$, where A is defined below Σ_i. Goals of this form are passed on to $PROVE_{\Delta_i}$ by line 4. $PROVE_{\Delta_i}$ must return *true* for all of these goals in order for $PROVE_{\Sigma_i}$ to return *true*.

Procedure $PROVE_{\Sigma_i}(\psi_0, DB_0)$:
    $GOALS \leftarrow \{(\psi_0, DB_0)\}$;
    do until $GOALS$ is empty
        choose some goal (ψ, DB) in $GOALS$;
        remove this goal from $GOALS$;
1.   if $\psi \in DB$ then continue;
2.   elseif $\psi = B[add : C]$ then $GOALS \leftarrow GOALS + \{(B, DB + C)\}$;

[11] These are the only type of formulas passed to $PROVE_{\Sigma_i}$ from $PROVE_{\Delta_{i+1}}$

3. elseif ψ is atomic and is defined in Σ_i then
choose a rule $\psi \leftarrow \psi_1, ..., \psi_m$ in Σ_i;
if such a rule exists
then $GOALS \leftarrow GOALS + \{(\psi_1, DB), ..., (\psi_m, DB)\}$;
else return *false*;
4. elseif $PROVE_{\Delta_i}(\psi, DB) = \mathit{false}$ then return *false*;
end do;
return *true*;
end procedure;

Each of the choice points in this procedure is non-deterministic, and if a proof of $R, DB \vdash \psi$ exists, then some sequence of choices will find it. More importantly, if a proof exists, then some sequence of choices of *polynomial length* will find it. This is true because recursion in Σ_i is linear (see appendix A). Thus, $PROVE_{\Sigma_i}$ runs in NP time relative to an oracle for $PROVE_{\Delta_i}$.

5.2.2 The Procedure $PROVE_{\Delta_i}$

Suppose that ψ is a formula of the form A or $\sim A$ where A is a ground atomic formula defined in Δ_i or below.[12] Then $PROVE_{\Delta_i}(\psi, DB) = \mathit{true}$ iff $R, DB \vdash \psi$.

Most of the inferences performed by $PROVE_{\Delta_i}$ are Horn inferences. Some of the rules in Δ_i may have hypothetical premises, but these are evaluated by invoking $PROVE_{\Sigma_{i-1}}$ as an oracle. The implementation of $PROVE_{\Delta_i}$ is therefore almost identical to the bottom-up method used to perform inference given a set of Horn rules with stratified negation [1].

Since negation in Δ_i is stratified, Δ_i can be partitioned into strata $\Delta_{i1}, ..., \Delta_{im_i}$ such that within each stratum, negation occurs only at the base level. The "perfect model" of Δ_i is then computed in a bottom-up fashion as follows: apply the rules in Δ_{i1} to DB until a fixpoint is reached; to this fixpoint, apply the rules in Δ_{i2} until a new fixpoint is reached; etc. After the rules in Δ_{im_i} have been applied, the resulting fixpoint is called the "perfect model" of Δ_i and DB [20].

The procedures below implement this idea. The first four procedures are exactly as they would be in the case of Horn rules. $PROVE_{\Delta_i}$ computes the "perfect model" by successively applying each stratum of Δ_i. It then determines whether ψ is true in this model. The remaining procedures support the computation of fixpoints. $LFP_i(\Delta, DB)$ applies the rules in Δ to DB until a fixpoint is reached; $T_i(\Delta, DB)$ applies each rule in Δ to DB exactly once; and $TEST_i(\psi, DB)$ determines whether DB satisfies ψ, where ψ is a rule premise.

Procedure $PROVE_{\Delta_i}(\psi, DB)$:

[12] These are the only type of formulas passed to $PROVE_{\Delta_i}$ by $PROVE_{\Sigma_i}$.

```
    do for j from 1 to m_i
        DB ← LFP_i(Δ_ij, DB);
    return TEST_i(ψ, DB);
    end procedure;

Procedure  LFP_i(Δ, DB):
    S ← T_i(Δ, DB);
    do until S = DB
        S ← DB;
        DB ← T_i(Δ, S);
    return DB;
    end procedure;

Procedure  T_i(Δ, DB):
    S ← DB;
    do for each rule  A ← ψ_1, ..., ψ_m  in Δ
        if  TEST_i(ψ_j, DB)  is true for each j from 1 to m
            then S ← S + {A};
    return S;
    end procedure;

Procedure  TEST_i(ψ, DB):
    if  ψ = ~ ψ'  then return not[TEST_i^0(ψ', DB)];
                  else return TEST_i^0(ψ, DB);
    end procedure;

Procedure  TEST_i^0(ψ, DB):
    if ψ ∈ DB then return true;
    elseif ψ is atomic and is defined in Δ_i then return false;
        else return PROVE_{Σ_{i-1}}(ψ, DB);
    end procedure;
```

Only the last procedure, $TEST_i^0(\psi, DB)$, distinguishes this algorithm from that of stratified Horn rules. Here, ψ is a rule premise with any negation signs removed. $TEST_i^0$ determines whether DB satisfies ψ. In the Horn-rule case, ψ would be atomic and would be satisfied iff $\psi \in DB$. In our case, however, ψ could be hypothetical or could involve predicates defined in a lower stratum. In these cases, $TEST_i^0$ invokes $PROVE_{\Sigma_{i-1}}$ as an oracle to evaluate ψ.

The corresponding algorithm for Horn rules with stratified negation runs in polynomial time (polynomial in the size of the data domain). Since $PROVE_{\Delta_i}$ is virtually identical, it runs in polynomial time relative to an oracle for $PROVE_{\Sigma_{i-1}}$.

6 Expressibility

In [5], hypothetical rulebases with non-linear recursion were examined. Two results were established: (i) that the graphs of such rulebases are in $PSPACE$, and (ii) that such rulebases can represent *any* typed, generic query whose graph is in $PSPACE$. The expressive power of hypothetical rules was thus established. In this section, we use the same techniques to extend these results to hypothetical rulebases with linear stratification. Section 5.2 showed that the graphs of rulebases with k strata are in Σ_k^P. This section shows the converse, that such rulebases can represent any typed, generic query whose graph is in Σ_k^P.

The proof relies on the encodings of oracle Turing-machines developed in the section 5.1. In this respect, it is similar to other expressibility proofs in the literature (e.g., [13, 22]). One difference, however, is that we do not assume that the data domain is linearly ordered. Ordered domains are used to simulate counters, which in turn, are used to simulate the movement of Turing-machine tape-heads. For hypothetical inference systems, however, ordering assumptions are unnecessary, for if there is no linear order on the domain, then one can be asserted hypothetically.

The difficulty with this approach is that a rulebase cannot select and assert a particular linear-order. Since there is no *a priori* ordering on the domain, there is no way for a rulebase to select one ordering over another. A rulebase can, however, assert all possible linear-orders, one after another, and simulate the oracle machines for each one. This technique works as long as the machine encodings are insensitive to the particular linear-order being used. This is the case for database queries which are *generic* [6, 7].

A query is generic iff it satisfies the following consistency criterion: if the constants in the database are renamed in a consistent way, then the constants in the answer to the query are renamed in the same way. In our machine encodings, changing the linear order is equivalent to renaming the database constants. Thus, if the machine computes a generic query, it does not matter which linear order is used. In this way, the consistency criterion is central to our ability to use unordered domains.

6.1 Database Queries

This section defines the notion of database query precisely. The main results regarding queries are then stated and reduced to a single lemma, which is proved in the next section. The definitions are essentially those of [6] and [7].

Definition 12 *(Relational Database) Let U be a countable set, called the universal data domain. A relational database DB of type $\overline{\alpha} = (\alpha_1, ..., \alpha_m)$ is a tuple $(D, R_1, ..., R_m)$ where D is a finite subset of U and R_i is an α_i-ary relation over D, i.e., $R_i \subseteq D^{\alpha_i}$. D is called the domain of DB, written dom(DB).*

In logical systems such as ours, a relational database is represented as a set of ground atomic formulas. U is a universal set of constant symbols, and for each relation R_i there is a predicate symbol P_i whose ground atomic formulas represent R_i.

Definition 13 *(Query) A generic database query of type $\overline{\alpha} \to \alpha_0$ is a partial function ψ which takes a database DB of type $\overline{\alpha}$ and returns a relation $\psi(DB)$ over $dom(DB)$ of arity α_0. In addition, ψ must satisfy the following consistency criterion: if DB' can be derived from DB by a renaming (i.e., a permutation) of the symbols in U,[13] then $\psi(DB')$ can be derived from $\psi(DB)$ by the same renaming.*

If one predicate symbol is reserved for the output relation, then a rulebase defines a typed database query. Not all such queries are generic however. Non-genericity means that some constant symbols are treated specially by the rulebase. If a set of hypothetical rules has no constant symbols, however, then no symbol is treated specially, and the resulting query is generic. We say that such rulebases are *constant free*.

Theorem 2 *Let $\mathcal{R}_k^{cf}$ be the set of hypothetical rulebases which are constant free and which have at most k levels of linear stratification. The set of relational queries expressed by $\mathcal{R}_k^{cf}$ is equal to the set of typed, generic queries whose graphs are in Σ_k^P.*

Corollary 1 *Let $\mathcal{R}^{cf}$ be the set of hypothetical rulebases which are constant free and which are linearly stratified. The set of relational queries expressed by $\mathcal{R}^{cf}$ is equal to the set of typed relational queries whose graphs are in Σ_*^P, where $\Sigma_*^P = \cup_{k \geq 1} \Sigma_k^P$.*

One direction is trivial. By theorem 1, the graph of any query in $\mathcal{R}_k^{cf}$ is in Σ_k^P; and by the above discussion, these queries are typed and generic. The other direction follows from the next lemma and its corollary. The lemma applies to yes/no queries, and the corollary generalizes it to typed queries. Whereas a typed query returns a set of tuples, a yes/no query simply returns *true* or *false*.

Lemma 2 *Suppose that ψ is a generic yes/no query, that is, a generic query of type $\overline{\alpha} \to 0$. Suppose also that the data-complexity of ψ is in Σ_k^P. Then there is a rulebase $R(\psi)$ in $\mathcal{R}_k^{cf}$ such that for all databases DB of type $\overline{\alpha}$,*

$$R(\psi), DB \vdash YES \quad \textit{iff} \quad \psi(DB) = \textit{true}$$

[13] In this case we say that DB and DB' are *isomorphic*.

where YES is a predicate symbol of arity zero.

Corollary 2 *Suppose that φ is a generic query of type $\overline{\alpha} \to \alpha_0$, and suppose that the graph of φ is in Σ_k^P. Then there is a rulebase $R(\varphi)$ in $\mathcal{R}_k^{cf}$ such that for all databases DB of type $\overline{\alpha}$,*

$$R(\varphi), DB \vdash OUT(\overline{x}) \quad \textit{iff} \quad \overline{x} \in \varphi(DB)$$

where OUT is a predicate symbol of arity α_0.

Proof: Let ψ be a yes/no query such that $\psi[DB + P_0(\overline{x})] = true$ iff $\overline{x} \in \varphi(DB)$, where P_0 is a predicate symbol not appearing in DB. Thus, if φ is of type $\alpha_1, ..., \alpha_m \to \alpha_0$, then ψ is of type $\alpha_0, \alpha_1, ..., \alpha_m \to 0$. Furthermore, since φ is generic and in Σ_k^P, so is ψ. Thus $R(\psi)$ exists by lemma 2. $R(\varphi)$ is constructed by adding to $R(\psi)$ the following rule:

$$OUT(\overline{x}) \leftarrow D_{\alpha_0}(\overline{x}),\ YES[add : P_0(\overline{x})].$$

where $D_{\alpha_0}(\overline{x})$ is an abbreviation for $D(x_1), ..., D(x_{\alpha_0})$. This rule generates all possible α_0-tuples $\overline{x}$ over $dom(DB)$. For each such tuple, $P_0(\overline{x})$ is hypothetically added to the database, so that $R(\psi)$ can be used to determine whether $\overline{x} \in \varphi(DB)$. That is,

$$\begin{aligned} & R(\varphi), DB \vdash OUT(\overline{x}) \\ \textit{iff} \quad & R(\psi), DB + P_0(\overline{x}) \vdash YES \\ \textit{iff} \quad & \psi[DB + P_0(\overline{x})] = true \\ \textit{iff} \quad & \overline{x} \in \varphi(DB) \end{aligned}$$

Note also that $R(\psi)$ and $R(\varphi)$ have the same number of strata. *QED*

6.2 Proving the Lemma

This section is devoted to the proof of lemma 2 and in particular, to the construction of $R(\psi)$.

By the premise of the lemma, the language $\{DB \mid \psi(DB) = true\}$ is in Σ_k^P. Thus there is a series of NP oracle-machines $M_k, ..., M_1$ which accepts this language. DB is the input to M_k, and each machine M_i uses M_{i-1} as its oracle. Section 5.1 showed how to encode this composite machine as a set of hypothetical rules which are constant free and which have k levels of linear stratification. To use this construction, two things must be done: *(i)* initialize the work tape of each of machine M_i, and *(ii)* encode a counter from 0 to $n^l - 1$, where n is the size of the data domain, and n^l is an upper bound on the amount of time used by any of the machines M_i.

6.2.1 Asserting a Linear Order

If there is a total linear order on the data domain, then a counter from 0 to $n^l - 1$ can be constructed by using predicates of arity l. For this reason, the assumption of a linearly-ordered domain is common in the literature [13, 22]), especially when expressibility results are established in terms of complexity classes, as in lemma 2. For hypothetical logics, however, this assumption is unnecessary, for if a linear order does not exist, then one can be asserted hypothetically.

The main difficulty with this is in choosing which linear order to assert. Since there is nothing special about any ordering, the rulebase $R(\psi)$ has no way of selecting one over another. One solution is for $R(\psi)$ to hypothetically assert every possible ordering, one at a time. This works because—as we shall see—the oracle-machine encodings are independent of which ordering is used. (A similar trick was used in implementing the query $EVEN$ in example 6.) In this way, no *a priori* domain ordering is needed and no distinguished ordering is selected.

The rules below use the technique of examples 6 and 7 to assert every possible ordering of the domain. Elements are selected from the domain D one at a time and inserted into a linear order. In particular, when the domain elements are selected in the order $a_1, a_2 ... a_n$, the following entries are hypothetically inserted into the database:

$$FIRST_1(a_1),\ NEXT_1(a_1, a_2),\ NEXT_1(a_2, a_3)\ ...\ NEXT_1(a_{n-1}, a_n),\ LAST_1(a_n).$$

This is in addition to the original database relations $P_0, P_1, ..., P_m$.

$$YES \leftarrow SELECT(x),\ ORDER(x)[add: FIRST_1(x)].$$
$$ORDER(x) \leftarrow SELECT(y),\ ORDER(y)[add: NEXT_1(x, y)].$$
$$ORDER(x) \leftarrow\ \sim SELECT(y),\ ACCEPT[add: LAST_1(x)].$$
$$SELECT(y) \leftarrow D(y),\ \sim SELECTED(y).$$
$$SELECTED(y) \leftarrow FIRST_1(y).$$
$$SELECTED(y) \leftarrow NEXT_1(x, y).$$

After inserting a linear order, the rules try to infer the atom $ACCEPT$, which in turn, invokes a simulation of the composite machine $M_k, ..., M_1$. Either $ACCEPT$ is inferred for all linear orders, or for none; thus $R(\psi), DB \vdash YES$ iff the composite machine accepts its input. Note that these rules are linear and that they reside in the top stratum of $R(\psi)$, i.e., the stratum which encodes M_k.

6.2.2 Representing the Oracle Machines

Each linear ordering of the data domain provides a way of counting from 0 to $n - 1$. By using predicates of arity l, this counter can be extended to go from 0 to $n^l - 1$. In particular, Horn rules can be used to define the following three predicates: $FIRST(\overline{x})$, $NEXT(\overline{x},\overline{y})$, and $LAST(\overline{y})$, where $\overline{x}$ and $\overline{y}$ are l-tuples. $FIRST(\overline{x})$ and $LAST(\overline{y})$ are true iff $\overline{x}$ and $\overline{y}$ represent the integers 0 and $n^l - 1$, respectively. $NEXT(\overline{x},\overline{y})$ is true iff $\overline{y}$ represents the integer $\overline{x}+1$. (See [5] for details).

With this counter, l-tuples can be used to represent n^l distinct points in time and n^l distinct positions on tape. The composite machine $M_k, ..., M_1$ can therefore be encoded much as was done in section 5.1. The following predicates represent machine M_i:

- $CELL_i^c(\overline{j},\overline{t})$: At time $\overline{t}$, the tape cell at position $\overline{j}$ contains symbol c.
- $CONTROL_i^q(\overline{j}_1,\overline{j}_2,\overline{t})$: At time $\overline{t}$, the finite control is in state q, the work head is over cell $\overline{j}_1$ of the work tape, and the oracle head is over cell $\overline{j}_2$ of the oracle tape.

The rulebase $R(\psi)$ must also compute the initial tape contents for each machine M_i. That is, the database DB must be encoded onto the work tape of M_k, and blanks must be encoded onto the work tapes of the other machines $M_{k-1}, ..., M_1$, as detailed in section 5.1. The latter is accomplished with $k-1$ rules. For each $i \leq k-1$, the following rule is added to the i^{th} stratum of $R(\psi)$:

$$CELL_i^b(\overline{j},\overline{t}) \leftarrow FIRST(\overline{t})$$

where b denotes a blank. This rule puts a b in every tape cell of M_i at time 0.

Encoding the database onto a tape is less straightforward. We exploit the fact that the database consists of a fixed number relations $P_0, P_1, ..., P_m$ of fixed arity $\alpha_0, \alpha_1, ..., \alpha_m$, resp. Each relation therefore has at most n^α entries, where n is the size of the data domain and α is the maximum of $\alpha_0, \alpha_1, ..., \alpha_m$. We divide an initial segment of the tape into blocks, each of size n^α, and we store a bit-map representation of P_i in the i^{th} block. That is, each tape cell in block i corresponds to a possible database entry $P_i(x_1, ..., x_{\alpha_i})$, and the cell contains a 1 iff $P_i(x_1, ..., x_{\alpha_i})$ is actually in the database. The use of negation-by-failure is crucial to setting a tape cell to 0 when the corresponding entry is *not* in the database.

It is straightforward, though somewhat tedious, to construct a set of Horn rules which set the bit-map cells to 0 or 1 and which make all other tape cells blank. In particular, for each symbol $c \in \{0, 1, b\}$, we can define a predicate $INIT^c(\overline{j})$ which is true iff c is the initial value of the tape cell at position $\overline{j}$. (See [5] for details). The work tape of machine M_k is then initialized by the following three rules, which reside in the k^{th} stratum of $R(\psi)$:

$$CELL_k^0(\overline{j},\overline{t}) \leftarrow INIT^0(\overline{j}),\ FIRST(\overline{t}).$$
$$CELL_k^1(\overline{j},\overline{t}) \leftarrow INIT^1(\overline{j}),\ FIRST(\overline{t}).$$
$$CELL_k^b(\overline{j},\overline{t}) \leftarrow INIT^b(\overline{j}),\ FIRST(\overline{t}).$$

6.2.3 Order Independence

Our machine representation depends on the existence of a total linear order on the data domain. If such an order is not provided *a priori*, then the rules of section 6.2.1 can assert one hypothetically. These rules assert every possible linear order, one after the other, simulating the composite machine $M_k, ..., M_1$ for each one. In most respects, the linear order is a mere implementation detail; it is used to simulate a counter, and the particular ordering used does not effect the computations that the machine performs. In one respect, however, this is not true: the encoding of the database onto M_k's work tape depends crucially on the linear order. In particular, different linear orders will result in different bit-map representations of the database.

Which of these bit maps actually represents the database? They all do. The composite machine either accepts all these bit maps, or it rejects them all. This is because our machine is special. It computes a generic yes/no query ψ. Since it is generic, ψ satisfies a *consistency criterion*: whether $\psi(DB)$ is *true* or *false* is not affected by renaming the constant symbols in DB; i.e., if the constant symbols in DB are renamed, then the value of $\psi(DB)$ does not change. It is not hard to see that changing the linear order on the data domain is equivalent to a renaming of the domain, at least as far as our composite machine is concerned. Thus, the different bit maps that the machine receives for each linear order represent isomorphic databases.

For example, suppose that ψ is a generic yes/no query which acts on databases consisting of a binary predicate P and a monadic predicate Q. Consider the particular database $DB = \{P(a,b),\ P(b,b),\ Q(b)\}$. In this case the data domain is $\{a,b\}$ and there are two possible linear orders on it, $a < b$ and $b < a$. The bit-map representations of this database under these two linear orders are shown in diagrams 1 and 2 below. In each diagram, the sequence of 1's and 0's represents the machine tape, and the entries beneath show the interpretation given to the tape cells.

1. Encoding the database $\{P(b,a),\ P(b,b),\ Q(b)\}$
   under the linear order $a < b$:

0	0	1	1	0	1
$P(a,a)$	$P(a,b)$	$P(b,a)$	$P(b,b)$	$Q(a)$	$Q(b)$

2. Encoding the database $\{P(b,a),\ P(b,b),\ Q(b)\}$
   under the linear order $b < a$:

1	1	0	0	1	0
$P(b,b)$	$P(b,a)$	$P(a,b)$	$P(a,a)$	$Q(b)$	$Q(a)$

3. Encoding the database $\{P(a,b),\ P(a,a),\ Q(a)\}$
   under the linear order $a < b$:

1	1	0	0	1	0
$P(a,a)$	$P(a,b)$	$P(b,a)$	$P(b,b)$	$Q(a)$	$Q(b)$

Diagrams 1 and 2 show clearly that changing the linear order changes the input to the machine. Diagrams 1 and 3 show that renaming the constant symbols changes the input in exactly the same way. In general, a re-ordering of the data domain is equivalent to a renaming. Thus, because our machine computes a generic query, it either accepts the database under all linear orderings, or it rejects the database under all linear orderings.

7 Summary

This paper has studied the data-complexity and expressive power of an extension of Horn-clause logic. This extension is centered on rules of the form $A \leftarrow B[add : C]$, which intuitively means, "infer A if inserting C allows the inference of B." In [5] it was shown that such rulebases are data-complete for $PSPACE$. The present paper has extended this line of research by developing syntactic restrictions with lower complexity. These restrictions are based on the ideas of *linear recursion* and *stratified negation.* (A rule is linear if recursion occurs through only one premise.) Although these two ideas are prominent in the theory of Horn-clause logic, they do not affect the data-complexity of Horn rulebases. This paper has shown, however, that negation and linearity have a profound effect on the complexity and expressibility of hypothetical rulebases.

First, a new notion of stratification was developed, in which linear recursion alternates with negation-by-failure. It was then shown that the data-complexity of a hypothetical rulebase depends on its degree of stratification. As the number of strata increases, data-complexity climbs the polynomial-time hierarchy. In particular, rulebases with k strata are data-complete for Σ_k^P.

The lower complexity bound was established by encoding cascaded oracle machines. Suppose that $M_k, ..., M_1$ is a sequence of NP oracle-machines in which each M_i uses M_{i-1} as its oracle. Then the composite machine $M_k, ..., M_1$ can be encoded as a hypothetical rulebase with k strata. Negation-by-failure is crucial to this encoding, for it allows the rules to detect those situations in which an oracle returns the answer "No". Without negation, only a single NP machine can be simulated, and complexity does not climb the polynomial-time hierarchy.

The upper complexity bound was established by defining a series of procedures $PROVE_k, ..., PROVE_1$, one for each stratum. Each $PROVE_i$ runs in NP-time, and uses $PROVE_{i-1}$ as an oracle. Each $PROVE_i$ is a mixed top-down/bottom-up proof procedure. The bottom-up component runs in polynomial time and is a simple variation of the familiar bottom-up procedure of Horn logic. The top-down component runs in NP time and generates proofs non-deterministically. Because of the restriction that hypothetical recursion be linear, $PROVE_i$ is guaranteed to find a proof of polynomial length, if a proof exists. In this way, the complexity of each stratum is in NP.

The direct encoding of Turing machines not only establishes a lower complexity bound, it also helps establish expressibility results. In [5], for instance, the encoding of alternating Turing machines was used to show that any database query computable in $PSPACE$ could be represented as a set of hypothetical rules. In the present paper, the encoding of oracle Turing machines admits a similar result: any database query whose graph is in Σ_k^P can be represented as a hypothetical rulebase having k levels of stratification.

Unlike other expressibility results in the literature, this result does not assume that the data domain is linearly ordered. Linearly ordered domains are typically used to simulate counters, which in turn, are used to simulate tape-head movements [13, 22]. The approach of this paper has been to start with unordered domains and to assert linear orders hypothetically. This technique works for all generic queries, that is, for all queries satisfying the consistency criterion of Chandra and Harel [6, 7].

Acknowledgements

The work of Thorne McCarty on the intuitionistic semantics of embedded implications was the original stimulus for this work. Thanks go to Tomasz Imielinski for helpful comments on the paper.

References

[1] K.R. Apt, H.A. Blair, and A. Walker. Towards a Theory of Declarative Knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 2, pages 89–148. Morgan Kaufmann, 1988.

[2] F. Bancilhon and R. Ramakrishnan. An Amateur's Introduction to Recursive Query Processing Strategies. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 16–52, Washington, D.C., May 28–30 1986.

[3] A.J. Bonner. A Logic for Hypothetical Reasoning. In *Proceedings of the Seventh National Conference on Artificial Intelligence*, pages 480–484, Saint Paul, Minnesota, August 21–26 1988. Published in expanded form as Technical Report TR-DCS-230, Department of Computer Science, Rutgers University, New Brunswick, NJ 08903.

[4] A.J. Bonner. Hypothetical Datalog: Complexity and Expressibility. In *Proceedings of the International Conference on Database Theory (ICDT)*, pages 144–160. Springer-Verlag, 1988. Published as volume 326 of *Lecture Notes in Computer Science*.

[5] A.J. Bonner. Hypothetical Datalog: Complexity and Expressibility. Technical Report DCS-TR-231, Department of Computer Science, Rutgers university, New Brunswick, NJ 08903, 1988. Also published in volume 76 (1990) of *Theoretical Computer Science (TCS)*, pages 3–51.

[6] A.K. Chandra and D. Harel. Computable Queries for Relational Databases. *Journal of Computer and System Sciences (JCSS)*, 21(2):156–178, 1980.

[7] A.K. Chandra and D. Harel. Structure and Complexity of Relational Queries. In *Proceedings of the Symposium on the Foundations of Computer Science (FOCS)*, pages 333–347, 1980.

[8] D.M. Gabbay. N-Prolog: an Extension of Prolog with Hypothetical Implications. II. Logical Foundations and Negation as Failure. *Journal of Logic Programming (JLP)*, 2(4):251–283, 1985.

[9] D.M. Gabbay and U. Reyle. N-Prolog: an Extension of Prolog with Hypothetical Implications. I. *Journal of Logic Programming (JLP)*, 1(4):319–355, 1984.

[10] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, New York, 1979.

[11] Ginsberg. Counterfactuals. *Artificial Intelligence*, 30(1):35–79, 1986.

[12] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[13] N. Immerman. Relational Queries Computable in Polynomial Time. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 147–152, 1982.

[14] R. Kowalski. *Logic for Problem Solving*. North-Holland, 1979.

[15] Sanjay Manchanda. *A Dynamic Logic Programming Language for Relational Updates*. PhD thesis, Department of Computer Science, State University of New York at Stony Brook, Stony Brook, New York, December 1987. Also published as Technical Report TR 88-2, Department of Computer Science, The University of Arizona, Tuscon, Arizona 85721, January, 1988.

[16] L.T. McCarty. Clausal Intuitionistic Logic. I. Fixed-Point Semantics. *Journal of Logic Programming (JLP)*, 5(1):1–31, 1988.

[17] L.T. McCarty. Clausal Intuitionistic Logic. II. Tableau Proof Procedures. *Journal of Logic Programming (JLP)*, 5(2):93–132, 1988.

[18] L.T. McCarty and N.S. Sridharan. The Representation of an Evolving System of Legal Concepts. II. Prototypes and Deformations. In *Proceedings of the Seventh IJCAI*, pages 246–253, 1981.

[19] D. Miller. A Logical Analysis of Modules in Logic Programming. In *Proceedings of the IEEE Symposium on Logic Programming*, pages 106–114, Sept 1986.

[20] T. Przymusinski. On the Declarative Semantics of Deductive Databases and Logic Programs. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 5, pages 193–216. Morgan Kaufmann, 1988.

[21] L.J. Stockmeyer. The Polynomial Time Hierarchy. *Theoretical Computer Science (TCS)*, 3(1):1–22, 1976.

[22] M. Vardi. The Complexity of Relational Query Languages. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, pages 137–146, 1982.

[23] D.S. Warren. Database Updates in Pure Prolog. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 244–253, 1984.

A Appendix

This appendix shows that if the procedure $PROVE_{\Sigma_i}(\psi, DB)$ returns *true*, then it generates a sequence of choices of polynomial length which returns *true*. Although $PROVE_{\Sigma_i}$ was described only for the propositional case, the predicate implementation is a straightforward generalization, and the proof below applies to the that case.

Central to the proof are two constants k_0 and k_i. k_0 is the maximum arity of all the predicate symbols in the database and the rulebase. The number of ground atomic formulas that can be constructed is thus $O(n^{k_0})$ where n is the number of constant symbols appearing in the data domain $dom(R, DB)$. k_i depends on the structure of recursion in Σ_i. In any rulebase, the set of predicates can be divided into equivalence classes in which the predicates in each class are mutually recursive [2]. k_i is the number of such classes in Σ_i.

The procedure $PROVE_{\Sigma_i}$ makes a sequence of non-determinisitic choices. For each sequence of choices, goals are removed from the goal set in a particular order, resulting in a sequence of goals (ψ_1, DB_1), (ψ_2, DB_2), ... Without loss of generality, we can focus on sequences in which each goal appears only once. Because recursion in Σ_i is linear, any such sequence can be of only polynomial length. In fact, the following theorem is the main result of this appendix.

Theorem 3 *Any sequence of goals generated by $PROVE_{\Sigma_i}$ without repetitions has length $O(n^{2k_ik_0})$.*

The proof of this theorem focusses on atomic goals, that is, on goals of the form (A, DB) where A is atomic and is defined in Σ_i. The length of any goal sequence generated by $PROVE_{\Sigma_i}$ is at most $m_0 + 1$ times the number of atomic goals. Here, m_0 is the maximum value of m over all rules $A \leftarrow \psi_1, ..., \psi_m$ in Σ_i. We prove that the number of atomic goals in any goal sequence is $O(n^{2k_ik_0})$.

The proof begins with several definitions.

Definition 14 *($\Rightarrow$)*

- $(A[add : B], DB) \Rightarrow (A, DB + B)$
- *Suppose that $A' \leftarrow \psi_1, ..., \psi_m$ is a rule in Σ_i, and that θ is a substitution such that $A = A'\theta$. Then $(A, DB) \Rightarrow (\psi_j\theta, DB)$ for each j.*

The symbol $\Rightarrow$ captures the notion of one goal spawning another. The two items in this definition correspond to lines 2 and 3 in the definition of $PROVE_{\Sigma_i}$. We let $\overset{*}{\Rightarrow}$ be the reflexive, transitive closure of $\Rightarrow$. Since we are focussing on atomic goals, it is convenient to ignore hypothetical goals and to speak of one atomic goal spawning another. For this, we extend the definition of $\Rightarrow$ by adding the following item to definition 14 (this does not affect the meaning of $\overset{*}{\Rightarrow}$):

- *If* $(A, DB) \Rightarrow (B[add : C], DB)$ *then* $(A, DB) \Rightarrow (B, DB + C)$

Definition 15 *A proof sequence is a sequence of atomic goals $G_0, G_1, G_2, ...$ such that no goal appears twice and $G_0 \overset{*}{\Rightarrow} G_j$ for each j.*

Note that $PROVE_{\Sigma_i}(A_0, DB_0)$ non-deterministically generates all possible proof sequences beginning with the goal (A_0, DB_0).

Definition 16 *If A_1 and A_2 are mutually recursive predicates, then they belong to the same equivalence class, and we write $A_1 \sim A_2$ and $(A_1, DB_1) \sim (A_2, DB_2)$.*

Lemma 3 *In any proof sequence G_0, G_1, G_2, ... there are $O(n^{2k_0})$ goals G_j such that $G_0 \sim G_j$.*

Proof: Because Σ_i is linear, there is at most one goal G_{j_1} in the proof sequence such that $G_0 \Rightarrow G_{j_1}$ and $G_0 \sim G_{j_1}$. Similarly, there is at most one goal G_{j_2} such that $G_{j_1} \Rightarrow G_{j_2}$ and $G_{j_1} \sim G_{j_2}$. Continuing in this way, we get a subsequence of atomic goals $G_0, G_{j_1}, G_{j_2}, ...$ In fact, this subsequence includes all the goals G_j in the proof sequence such that $G_0 \sim G_j$. We show that the length of this subsequence is $O(n^{2k_0})$.

If (A, DB) and (A', DB') are two consecutive goals in this subsequence, then $(A, DB) \Rightarrow (A', DB')$, so $DB \subseteq DB'$. Thus, the subsequence can be divided into segments such that within each segment, the database is the same, and between segments, it increases. Each segment has the form $(A_1, DB), (A_2, DB), (A_3, DB)...$ where the atomic formulas A_i are distinct. Thus the length of a segment is no greater than the number of possible atomic formulas, that is, $O(n^{k_0})$. In going from one segment to the next, however, the database increases, that is, a ground atomic formula is added to it. This can only happen $O(n^{k_0})$ times, however, before the database saturates and equals the set of all possible ground atomic formulas. Thus, the subsequence contains $O(n^{k_0})$ segments each of length $O(n^{k_0})$. Its total length is therefore $O(n^{2k_0})$. *QED*

Each of the $O(n^{2k_0})$ goals mentioned in this lemma are in the same equivalence class as G_0. Each of them, however, may spawn other goals which are not. This idea leads to a hierarchy of goals starting at G_0.

Definition 17 *(Progeny) Suppose that G_0, G_1 and G_2 are atomic goals. Then*

- *If $G_0 \stackrel{*}{\Rightarrow} G_1$ and $G_0 \sim G_1$ then G_1 is a child (or a progenus of degree 1) of G_0*
- *Suppose that G_1 is a progenus of G_0 of degree d. If $G_1 \Rightarrow G_2$ and $G_1 \not\sim G_2$, then every child of G_2 is a progenus of G_0 of degree $d+1$.*

The concept of children reflects recursion within an equivalence class. Each time a non-recursive call is made to another equivalence class,[14] however, the degree increases by one. Thus, no goal can have degree greater than k_i, the number of equivalence classes in Σ_i. Note also that in a proof sequence G_0, G_1, G_2, ... every goal G_j is a progenus of G_0 to some degree.

Lemma 4 *In any proof sequence, the first goal G_0 has $O(n^{2dk_0})$ progeny of degree d.*

Proof: (By induction on j). The base case ($d = 1$) is established by lemma 3. Suppose, then, that the lemma is true for d. We constuct all possible progeny of G_0 of degree $d+1$ as follows. Let G_1 be a progenus of G_0 of degree d which appears in the proof sequence. G_1 spawns at most m_0 atomic subgoals, and each subgoal G_2 will have $O(n^{2k_0})$ children in the proof sequence, by lemma 3. Thus, in the proof sequence, each of the $O(n^{2dk_0})$ progeny of degree d has $O(m_0)$ subgoals, each having $O(n^{2k_0})$ children. Thus, G_0 has at most $O(n^{2dk_0}) \times O(m_0) \times O(n^{2k_0})$ progeny of degree $d+1$, that is, $O(n^{2(d+1)k_0})$ since m_0 is a constant. *QED*

Corollary 3 *Any proof sequence is of length $O(n^{2k_ik_0})$.*

[14] as represented by the item "$G_1 \Rightarrow G_2$ and $G_1 \not\sim G_2$" in the definition.